MISSION CONTROL v8.0

Week 8 Workshop: Files, Exceptions and Debugging

Building a persistent, safer and easier-to-debug Mission Control Panel

Dr Lei Wang (Griffith University)

Mission briefing
Mission Control can already monitor sensor temperatures, classify readings and analyse its history.

But it has a serious limitation: when the program shuts down, its in-memory history disappears.

This week, we upgrade the system so that it can remember data between runs, save reports, handle expected problems safely, and help us investigate faults systematically.

What You Are Building

We are continuing the same Mission Control project from Week 7. The goal is not to replace the existing program, but to improve it step by step using the new ideas from this week's lecture.

Mission Control v8.0 introduces persistent text files, file reading and writing, exception handling, defensive programming and systematic debugging.

The result is a program that can keep information after shutdown and respond more gracefully when something goes wrong.

What changes from v7.0?

v7.0	v8.0
History exists only in a Python list	History is also stored in a text file
Data disappears when the program ends	Previous readings are loaded at startup
Input conversion can crash on bad input	Invalid numeric input is handled safely
Reports are displayed only	Reports can be saved to a file
No file inspection command	A preview command reads the history file
Limited recovery from file problems	Expected file/data problems are handled
Debugging is mainly manual	Debugging becomes a deliberate test-and-inspect process

Learning Goals

- Open text files using r, w, a and x modes.
- Use with open(...) so files are closed automatically.
- Read files line by line and use readlines() when a list of lines is useful.
- Combine file processing with string operations such as strip(), split(), and rstrip().

- Write text with write() and understand that write() does not automatically add a newline.
- Use try/except for expected run-time problems, especially ValueError, FileNotFoundError and FileExistsError.
- Recognise how else and finally work in exception handling.
- Understand defensive programming and programming by contract.
- Use a systematic debugging process: understand, isolate, test, inspect, fix, and test again.
- Use print statements and PyCharm breakpoints to inspect program state rather than guessing.

Files You Need for This Workshop

The workshop package contains several files. Each one has a specific role. Keep the files together in the same folder while you work.

File	Purpose	How we use it
mission_control_v8.py	Main Mission Control program	Build, run and debug the program
mission_history.txt	Normal persistent sensor history	Load, append, preview and reset data
mission_report.txt	Saved mission report	Save or create a report
mission_history_malformed.txt	Deliberately problematic history data	Test defensive programming and exceptions
mission_history_edge_cases.txt	Boundary-value test data	Test SAFE/WARNING/CRITICAL boundaries
README.txt	Support-file instructions	Setup and testing guidance

Keep mission_control_v8.py and the supplied text files in the same folder.

The program uses simple filenames such as mission_history.txt, so Python looks for these files in the program's working directory.

Before You Start

- Open mission_control_v8.py in PyCharm.
- Make sure the supplied .txt files are in the same folder.
- Run the program before changing anything.
- Enter a mission name and destination when prompted.
- Try history, report, recent, warning, critical, top3, mission and preview.
- Do not delete or edit the supplied normal history file yet; we will use it for testing.

You should see that Mission Control can recover previously stored readings. That is our first clue that the program now has persistent memory.

How the History File Stores Data

A text file is simply text. To store structured sensor records, we need a consistent format. Mission Control stores one reading per line:

```
1|23.5|SAFE
2|81.0|WARNING
3|105.0|CRITICAL
```

The vertical bar character, |, separates the three fields. When we read a line, split("|") can turn the text back into separate values.

Field	Example	Meaning
Sensor number	1	Which sensor produced the reading
Temperature	23.5	The measured temperature
Status	SAFE	Classification produced by Mission Control

Upgrade 1 - Give Mission Control a Persistent Memory

The first upgrade is to load previous readings when the program starts. The existing reading_history list remains the program's working data, but the text file becomes its persistent storage.

Step 1: Define the filenames

```
HISTORY_FILE = "mission_history.txt"
REPORT_FILE = "mission_report.txt"
```

Constants give the program one clear place to store the filenames.

Step 2: Load the history

The loader follows a useful file-processing pattern:

```
Open -> read a line -> clean it -> split it -> validate it
     -> convert values -> create a tuple -> store it
```

The actual load_history() function in the final program also handles empty lines, malformed records, invalid numeric values, invalid statuses and a missing history file.

> FileNotFoundError is expected when Mission Control is run for the first time and no history file exists. Instead of crashing, the program starts with an empty history.

Step 3: Test with the supplied history file

- Run Mission Control with mission_history.txt present.
- Check the number of historical readings loaded.
- Run history and compare the displayed readings with the file.
- Run report and check that the statistics include the loaded data.
- Run preview to inspect the first and last records in the file.

Upgrade 2 - Save New Sensor Readings

Loading gives Mission Control a memory at startup. Next, we need to keep new readings after each scan.

This is where file mode 'a' becomes important. Append mode adds new text to the end of an existing file.

```
def append_history(filename, readings):
    with open(filename, "a") as file:
        for sensor_number, temperature, status in readings:
            file.write(
                f"{sensor_number}|{temperature}|{status}\n"
            )
```

Notice the newline character at the end of each record. write() does exactly what we ask it to do; it does not automatically move to a new line.

Memory versus persistent storage

Location	Example	What happens when the program ends?
Python memory	reading_history	The data disappears
Text file	mission_history.txt	The data remains

After a successful scan, Mission Control updates both: it extends reading_history and appends the new records to the file.

File Modes: Choose the Operation Carefully

Mode	What it does	Mission Control example
r	Reads an existing file	Load history or preview a file
w	Writes from the beginning and replaces existing contents	Save a fresh report or reset history
a	Adds new data to the end	Append new sensor readings
x	Creates a new file and fails if it already exists	Create a new report safely

The mode is not just a letter to memorise. Ask what you want to happen to the existing contents. Do you want to read them, replace them, add to them, or create a file only if it is new?

Upgrade 3 - Make Sensor Input Safer

The user controls the input, so the program cannot assume that every response can be converted to a number. For example, float('hot') raises ValueError.

We handle this expected problem with try/except and repeat the input until a valid number is supplied.

```
while True:
    try:
        temperature = float(
```

```
            input(f"Sensor {sensor_number} temperature: ")
        )
        break
    except ValueError:
        print("Invalid temperature. Please enter a number.")
```

The risky operation is inside try. If it succeeds, break leaves the validation loop. If it fails, except provides a useful recovery message.

Test the Temperature Boundaries

The supplied mission_history_edge_cases.txt is designed for deliberate testing. The classification rules are:

```
temperature < 0       -> INVALID
temperature >= 100    -> CRITICAL
temperature > 80      -> WARNING
otherwise             -> SAFE
```

Before running the test, predict the status of these values:

Temperature	Your prediction
0	
80	
80.1	
99.9	
100	

This is an example of boundary-value testing. Values close to a decision boundary are especially useful because they can reveal incorrect comparison operators such as >= versus >.

Upgrade 4 - Save a Mission Report

Mission Control already calculates a report. We now make that report persistent too.

```
def save_report(filename, report_lines):
    with open(filename, "w") as file:
        file.write("\n".join(report_lines))
        file.write("\n")
```

Mode 'w' is appropriate here because the saved report should represent the current report, not be mixed with an older report.

Try this sequence:

- Run report.
- Run save.
- Open mission_report.txt and inspect it.
- Perform another scan.
- Run save again and check that the report is replaced with the new current report.

Upgrade 5 - Create a New Report Safely

Sometimes we want to create a file only if it does not already exist. Mode 'x' is designed for this.

```
def create_new_report(filename, report_lines):
    try:
        with open(filename, "x") as file:
            file.write("\n".join(report_lines))
            file.write("\n")
        print(f"New report created: {filename}")
    except FileExistsError:
        print(
            f"Report '{filename}' already exists. "
            "Use the save command to replace it."
        )
```

Run newreport twice. The first attempt should create the file. The second should be handled by FileExistsError instead of crashing.

Upgrade 6 - Preview a File

Mission Control includes a preview command so we can inspect the persistent history without loading it into the main history list.

```
with open(filename, "r") as file:
    lines = file.readlines()

print(f"Lines in file: {len(lines)}")

for line in lines[:2]:
    print(line.rstrip())

if len(lines) > 2:
    for line in lines[-2:]:
        print(line.rstrip())
```

readlines() gives us a list of lines, so we can use slicing such as lines[:2] and lines[-2:]. The newline is part of each line read from a text file, so rstrip() removes it before display.

Another important reading pattern

```
with open(filename, "r") as file:
    for line in file:
        line = line.strip()
        # process line
```

This is particularly useful when a file contains many lines and we want to process each line once.

Upgrade 7 - Reset Persistent History

The reset command uses 'w' to clear the history file. The Python list is also reset so memory and storage agree.

```
def reset_history(filename):
    with open(filename, "w") as file:
        file.write("")

    print(f"History file '{filename}' has been cleared.")
```

Be careful with 'w': Opening a file in write mode replaces its contents. That makes it useful for reset and report

saving, but dangerous if used when you actually meant to append.

Upgrade 8 - Use Standard Error Output

Python provides standard file handles including sys.stdin, sys.stdout and sys.stderr. Mission Control uses stderr for diagnostic warnings about bad history records.

```
import sys

print("Warning: skipped malformed line", file=sys.stderr)
```

The important idea is that normal program output and diagnostic/error output can be separated.

Testing with the Supplied Problem Files

Test A - Normal mission history

Use mission_history.txt as the normal starting dataset. Verify that the program loads the existing readings and that history, report, recent, status filtering and top3 work as expected.

Test B - Malformed history

Now use mission_history_malformed.txt as a deliberately damaged dataset. It contains examples of an invalid temperature, an incorrectly formatted line, an invalid status and an empty line.

Your goal is not simply to make every line valid. Your goal is to observe whether Mission Control can reject bad records while continuing to use the valid ones.

Problem	Relevant Week 8 idea
Missing file	FileNotFoundError
Non-numeric temperature	ValueError
Wrong number of fields	Validation / defensive programming
Invalid status	Validation / ValueError
Empty line	strip() and simple conditional logic

Test C - Boundary values

Use mission_history_edge_cases.txt to test the classification boundaries. Predict the results first, then run the program.

Errors: Three Different Kinds of Problems

Type	Meaning	Example
Syntax error	Python cannot parse the program	Missing colon

Logical / semantic error	The program runs but does the wrong thing	Incorrect report calculation
Run-time error	A problem occurs while the program is running	Missing file or invalid conversion

A program that runs without a syntax error is not necessarily correct. Testing and debugging are needed to discover logical errors and unexpected run-time behaviour.

Exceptions: try, except, else and finally

```
try:
    # risky operation
except SomeException:
    # recovery
else:
    # runs when try succeeds
finally:
    # runs whether an exception occurred or not
```

Mission Control uses try/except where operations can reasonably fail at run time. else is useful for code that should run only after the risky operation succeeds. finally is useful for cleanup that should happen regardless of whether an exception occurred.

Important exceptions from this week include:

- ValueError - a value has the wrong form, such as float('hot').
- NameError - a name is used before it has been defined.
- ZeroDivisionError - a calculation attempts to divide by zero.
- FileNotFoundError - a requested file cannot be found.
- FileExistsError - an operation such as x mode encounters an existing file.

Do not use exceptions for everything
If a simple condition can be checked directly with if, use the condition. Use exception handling when an operation can reasonably fail during execution.

Defensive Programming and Programming by Contract

Defensive programming means anticipating reasonably expected problems at the boundaries of a program, such as invalid user input or damaged file data.

Programming by contract gives functions clear expectations. A precondition states what should be true before a function is used. A postcondition states what the function guarantees after successful execution.

```
def middle_char(text):
    # Precondition: text is not empty.
    # Postcondition: returns the middle character.
    return text[len(text) // 2]
```

In Mission Control, the history loader effectively applies a contract to file records: a valid record must have the expected fields, valid numeric values and a recognised status before it enters reading_history.

Debugging Mission Control Systematically

Debugging is not random code editing. Use evidence and a repeatable process:

```
Understand
    ->
Isolate
    ->
Test
    ->
Inspect
    ->
Fix
    ->
Test again
```

Technique 1 - Test as you code

After adding one feature, run the program before adding the next feature. If something breaks, you know the problem is likely close to the most recent change.

Technique 2 - Inspect program state

```
print("DEBUG readings:", reading_history)
print("DEBUG parts:", parts)
print("DEBUG status:", status)
```

Temporary debugging output can reveal what the program actually knows at a particular point.

Technique 3 - Read the traceback

When Python reports an exception, start near the bottom of the traceback. Identify the exception type, look at the reported line, and then inspect the values that reached that line.

Technique 4 - Isolate the defect

Reduce the problem to the smallest section that still fails. A smaller failing example is much easier to understand.

Technique 5 - Use a PyCharm breakpoint

- Place a breakpoint inside the suspicious function or loop.
- Run the program in debug mode.
- Inspect variables when execution pauses.
- Step through the program one statement at a time.
- Find the first point where the program state becomes different from what you expected.

A Complete Mission Control Test Mission

Use this sequence to test the finished system in a controlled way.

Stage	Action	What you should learn
1. Start	Run with mission_history.txt.	Files can provide persistent starting data.
2. Inspect	Run history and preview.	The list and file can represent the same data.

3. Analyse	Run report, recent, warning and top3.	Existing Week 7 features still work.
4. Scan	Enter valid temperatures.	New readings enter memory.
5. Persist	Run shutdown, then restart.	New readings survive the program ending.
6. Recover	Enter abc as a temperature.	ValueError can be handled safely.
7. Save	Run save and inspect mission_report.txt.	w mode replaces report contents.
8. Create	Run newreport twice.	x mode protects an existing file.
9. Diagnose	Test malformed history data.	Bad records can be rejected without losing valid data.
10. Reset	Run reset and then history.	Persistent storage and memory can be cleared together.

Common Mistakes

Mistake	Why it causes trouble	Better approach
Using 'w' instead of 'a'	Existing history is replaced.	Use 'a' when adding new records.
Forgetting \n in write()	Records can run together.	Write a newline after each record.
Forgetting to close files	Resources may remain open.	Prefer with open(...).
Treating file fields as numbers automatically	split() produces strings.	Convert numeric fields explicitly.
Using a bare except	Unexpected errors can be hidden.	Catch specific expected exceptions.
Using try/except for simple conditions	Code becomes harder to read.	Use if when the condition is directly testable.
Changing many things before testing	The source of a defect becomes unclear.	Make one small change and test.
Ignoring malformed records	One bad line can disrupt loading.	Validate and skip invalid records.
Confusing memory with storage	Data may appear to disappear unexpectedly.	Remember that lists are temporary; files persist.

Challenge Tasks

- Add a command that counts the number of readings stored in the persistent history file.
- Add a command that searches the persistent file directly.

- Create a separate alert file containing only WARNING and CRITICAL readings.
- Add the mission name and destination to the saved report.
- Add a timestamp field to each stored reading and update the loader.
- Use a PyCharm breakpoint inside load_history() and trace one malformed record from reading to rejection.

Week 8 Checklist

- I can explain the difference between temporary program memory and persistent file storage.
- I know what r, w, a and x do.
- I can use with open(...) to work safely with files.
- I can read a file using read(), readline(), readlines(), or for line in file.
- I can explain why write() needs \n when I want separate lines.
- I can use strip(), split() and rstrip() when processing file text.
- I can explain FileNotFoundError, FileExistsError and ValueError.
- I can use try/except for appropriate run-time problems.
- I understand the purpose of else and finally in exception handling.
- I can describe defensive programming and programming by contract.
- I can use a traceback, print statements and breakpoints to investigate a bug.
- I can test boundary values rather than relying only on ordinary examples.
- I can explain why testing after each small change makes debugging easier.

Final Mission Control v8.0 Code

The complete reference implementation is reproduced below. The code itself has been kept unchanged. Build the program incrementally first, then use this section to check your work and debug your implementation.

```
# ============================================================
# MISSION CONTROL v8.0
# Week 8: Files, Exceptions and Debugging
# Based on Mission Control v7.0
# ============================================================

import sys

HISTORY_FILE = "mission_history.txt"
REPORT_FILE = "mission_report.txt"


def show_header(title):
    print("=" * 60)
    print(title.center(60))
    print("=" * 60)


def classify_temperature(temperature):
    if temperature < 0:
        return "INVALID"
    elif temperature >= 100:
        return "CRITICAL"
    elif temperature > 80:
        return "WARNING"
    else:
        return "SAFE"


def format_reading(reading):
    sensor_number, temperature, status = reading
    return f"Sensor {sensor_number:<2} | {temperature:>6.1f} C | {status:<8}"


def load_history(filename):
```

```
    reading_history = []

    try:
        with open(filename, "r") as file:
            for line in file:
                line = line.strip()

                if line == "":
                    continue

                parts = line.split("|")

                if len(parts) != 3:
                    print(
                        f"Warning: skipped malformed line: {line}",
                        file=sys.stderr
                    )
                    continue

                try:
                    sensor_number = int(parts[0])
                    temperature = float(parts[1])
                    status = parts[2]

                    if status not in ("SAFE", "WARNING", "CRITICAL"):
                        raise ValueError("invalid status")

                    reading = (sensor_number, temperature, status)
                    reading_history.append(reading)

                except ValueError:
                    print(
                        f"Warning: skipped invalid reading: {line}",
                        file=sys.stderr
                    )

    except FileNotFoundError:
        print(
            f"No existing history file found. Starting a new mission log."
        )

    return reading_history


def append_history(filename, readings):
    with open(filename, "a") as file:
        for sensor_number, temperature, status in readings:
            file.write(
                f"{sensor_number}|{temperature}|{status}\n"
            )


def save_report(filename, report_lines):
    with open(filename, "w") as file:
        file.write("\n".join(report_lines))
        file.write("\n")


def create_new_report(filename, report_lines):
    try:
        with open(filename, "x") as file:
            file.write("\n".join(report_lines))
            file.write("\n")
        print(f"New report created: {filename}")
    except FileExistsError:
        print(
            f"Report '{filename}' already exists. "
            "Use the save command to replace it."
        )


def scan_sensors():
    scan_readings = []
    safe_count = 0
    warning_count = 0

    for sensor_number in range(1, 6):
        while True:
            try:
                temperature = float(
                    input(f"Sensor {sensor_number} temperature: ")
                )
```

```
                break
            except ValueError:
                print("Invalid temperature. Please enter a number.")

        status = classify_temperature(temperature)

        if status == "INVALID":
            print("Invalid reading - skipped.")
            continue

        reading = (sensor_number, temperature, status)
        scan_readings.append(reading)

        if status == "CRITICAL":
            print("CRITICAL - scan stopped")
            warning_count += 1
            break
        elif status == "WARNING":
            print("WARNING")
            warning_count += 1
        else:
            print("SAFE")
            safe_count += 1

    return scan_readings, safe_count, warning_count


def show_report(reading_history):
    print("\n--- CURRENT REPORT ---")
    total_valid = len(reading_history)
    print(f"Valid readings: {total_valid}")

    if not reading_history:
        print("No valid sensor data yet.")
        return []

    temperatures = [reading[1] for reading in reading_history]
    statuses = [reading[2] for reading in reading_history]

    safe_count = statuses.count("SAFE")
    warning_count = statuses.count("WARNING")
    critical_count = statuses.count("CRITICAL")

    average = sum(temperatures) / len(temperatures)

    report_lines = [
        "--- CURRENT REPORT ---",
        f"Valid readings: {total_valid}",
        f"Safe readings: {safe_count}",
        f"Warnings: {warning_count}",
        f"Critical readings: {critical_count}",
        f"Average: {average:.1f} C",
        f"Highest: {max(temperatures):.1f} C",
        f"Lowest: {min(temperatures):.1f} C"
    ]

    for line in report_lines[1:]:
        print(line)

    return report_lines


def show_history(reading_history):
    print("\n--- READING HISTORY ---")

    if not reading_history:
        print("No valid sensor readings recorded.")
        return

    lines = []
    for reading in reading_history:
        lines.append(format_reading(reading))

    print("\n".join(lines))


def show_recent(reading_history):
    print("\n--- RECENT READINGS ---")
    recent_readings = reading_history[-3:]

    if not recent_readings:
        print("No valid sensor readings recorded.")
```

```
        return

    for reading in recent_readings:
        print(format_reading(reading))


def show_status(reading_history, required_status):
    matching_readings = [
        reading
        for reading in reading_history
        if reading[2] == required_status
    ]

    print(f"\n--- {required_status} READINGS ---")

    if not matching_readings:
        print(f"No {required_status} readings recorded.")
        return

    for reading in matching_readings:
        print(format_reading(reading))


def show_highest_three(reading_history):
    print("\n--- HIGHEST THREE TEMPERATURES ---")

    if not reading_history:
        print("No valid sensor readings recorded.")
        return

    temperatures = [reading[1] for reading in reading_history]
    temperatures.sort()

    for temperature in temperatures[-3:][::-1]:
        print(f"{temperature:.1f} C")


def show_mission(mission_name, destination):
    print("\n--- MISSION INFORMATION ---")
    print(f"Mission: {mission_name}")
    print(f"Destination: {destination}")

    if len(mission_name) >= 3:
        print(f"Mission code: {mission_name[:3].upper()}")
    else:
        print("Mission code: " + mission_name.upper())


def search_history(reading_history):
    keyword = input("Search text: ").strip().lower()

    if keyword == "":
        print("Search text cannot be empty.")
        return

    matches = []

    for reading in reading_history:
        line = format_reading(reading)

        if keyword in line.lower():
            matches.append(line)

    print("\n--- SEARCH RESULTS ---")

    if not matches:
        print("No matching readings found.")
    else:
        print("\n".join(matches))


def status_command(reading_history):
    status = input(
        "Enter status (SAFE/WARNING/CRITICAL): "
    ).strip().upper()

    if status in ("SAFE", "WARNING", "CRITICAL"):
        show_status(reading_history, status)
    else:
        print("Invalid status.")
```

```
def build_log_lines(reading_history):
    lines = []

    for reading in reading_history:
        lines.append(format_reading(reading))

    return lines


def show_log(reading_history):
    print("\n--- MISSION LOG ---")

    if not reading_history:
        print("No valid sensor readings recorded.")
        return

    lines = build_log_lines(reading_history)
    log = "\n".join(lines)

    print(log)
    print(f"\nLog contains {len(lines)} readings.")


def save_current_report(reading_history):
    report_lines = show_report(reading_history)

    if not report_lines:
        return

    save_report(REPORT_FILE, report_lines)
    print(f"Report saved to {REPORT_FILE}.")


def create_report_file(reading_history):
    report_lines = show_report(reading_history)

    if not report_lines:
        return

    create_new_report(REPORT_FILE, report_lines)


def show_file_preview(filename):
    print(f"\n--- FILE PREVIEW: {filename} ---")

    try:
        with open(filename, "r") as file:
            lines = file.readlines()

        if not lines:
            print("File is empty.")
            return

        print(f"Lines in file: {len(lines)}")
        print("First two lines:")

        for line in lines[:2]:
            print(line.rstrip())

        if len(lines) > 2:
            print("Last two lines:")
            for line in lines[-2:]:
                print(line.rstrip())

    except FileNotFoundError:
        print(f"File '{filename}' was not found.")


def reset_history(filename):
    with open(filename, "w") as file:
        file.write("")

    print(f"History file '{filename}' has been cleared.")


def show_help():
    command_text = (
        "scan report history recent safe warning critical "
        "top3 mission search status log save newreport preview "
        "reset help shutdown"
    )
```

```
    commands = command_text.split()

    print("\n--- AVAILABLE COMMANDS ---")
    print(" | ".join(commands))


def run_mission_control():
    reading_history = load_history(HISTORY_FILE)

    mission_name = input("Mission name: ").strip().title()
    destination = input("Destination: ").strip().title()

    show_header(f"MISSION CONTROL v8.0 — {mission_name}")
    print(f"Destination: {destination}")
    print(f"Loaded {len(reading_history)} historical readings.")

    while True:
        command = input(
            "\nCommand [scan/report/history/recent/safe/warning/"
            "critical/top3/mission/search/status/log/save/newreport/"
            "preview/reset/help/shutdown]: "
        ).strip().lower()

        if command == "shutdown":
            print("Shutting down Mission Control...")
            break

        if command == "report":
            show_report(reading_history)
            continue

        if command == "history":
            show_history(reading_history)
            continue

        if command == "recent":
            show_recent(reading_history)
            continue

        if command == "safe":
            show_status(reading_history, "SAFE")
            continue

        if command == "warning":
            show_status(reading_history, "WARNING")
            continue

        if command == "critical":
            show_status(reading_history, "CRITICAL")
            continue

        if command == "top3":
            show_highest_three(reading_history)
            continue

        if command == "mission":
            show_mission(mission_name, destination)
            continue

        if command == "search":
            search_history(reading_history)
            continue

        if command == "status":
            status_command(reading_history)
            continue

        if command == "log":
            show_log(reading_history)
            continue

        if command == "save":
            save_current_report(reading_history)
            continue

        if command == "newreport":
            create_report_file(reading_history)
            continue

        if command == "preview":
            show_file_preview(HISTORY_FILE)
            continue
```

```
        if command == "reset":
            reset_history(HISTORY_FILE)
            reading_history = []
            continue

        if command == "help":
            show_help()
            continue

        if command != "scan":
            print("Unknown command.")
            continue

        scan_readings, scan_safe, scan_warning = scan_sensors()

        if scan_readings:
            reading_history.extend(scan_readings)
            append_history(HISTORY_FILE, scan_readings)

        print(
            f"Scan complete: {len(scan_readings)} valid readings, "
            f"{scan_safe} safe, {scan_warning} warning/critical."
        )

    print("=" * 60)
    print("SESSION ENDED".center(60))
    print("=" * 60)


run_mission_control()
```

Final Mission Briefing

Mission Control v8.0 is more than an extension of the previous program. It introduces a new way of thinking about programs: software interacts with persistent data and with users, so it must be prepared for missing files, invalid input and unexpected run-time situations.

The most important habit from this week is not a particular file mode or exception name. It is the engineering process: build a small piece, test it, inspect what happened, isolate problems, fix them, and test again. Those habits will continue to be useful as Mission Control becomes more sophisticated.

Appendix - Suggested File Setup

For the workshop, place the supplied files together:

```
Mission_Control_v8/
|
+-- mission_control_v8.py
+-- mission_history.txt
+-- mission_report.txt
+-- mission_history_malformed.txt
+-- mission_history_edge_cases.txt
+-- README.txt
```

Keep the normal history file unchanged until the relevant testing activity. The malformed and edge-case files are intentionally provided for testing and discussion.